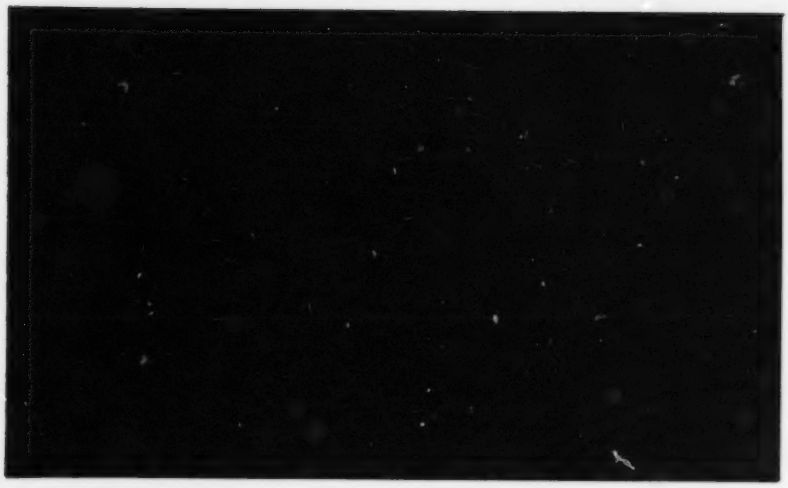
CENTRE DE RECHERCHE SUR LES TRANSPORTS
(C.R.T.)
CENTRE FOR RESEARCH ON TRANSPORTATION

UNIVERSITÉ DE MONTRÉAL
ÉCOLE DES HAUTES ÉTUDES COMMERCIALES
ÉCOLE POLYTECHNIQUE

CENTRE DE RECHERCHE SUR LES TRANSPORTS
UNIVERSITÉ DE MONTRÉAL
C.P. 6128, SUCCURSALE A
MONTRÉAL
CANADA H3C 3J7

TÉLÉPHONE: (514) 343-7575
TÉLEX: 05-24146 BIBPOLYTEC
EMAIL: crt@crt.umontreal.ca
TÉLÉCOPIEUR (FAX): (514) 343-7121



92-04604/

CRT-817

Implementing an Insertion Heuristic for Vehicle Routing on Parallel Hardware

by

Christian Foisy
Jean-Yves Potvin

Centre de recherche sur les transports, Université de Montréal, C.P. 6128, succursale A, Montréal, Québec, Canada H3C 3J7

Centre de recherche sur les transports — Publication #817

February 1992

Dépôt légal – 1er trimestre 1992

Résumé. Dans cet article, nous discutons de l'implantation sur une architecture parallèle d'une heuristique d'insertion pour le problème de tournées de véhicules avec fenêtres horaires. A l'aide d'un ensemble de problèmes tests, nous démontrons les gains en temps qui peuvent être obtenus en distribuant le travail entre plusieurs processeurs. Nous décrivons également deux procédures permettant de réduire de façon importante les coûts de communication entre les processeurs ainsi que les périodes d'inactivité.

Mots Clés. Calcul Parallèle, Tournées de Véhicules, Heuristique d'Insertion, Transputer.

Abstract. In this paper, we report about the implementation of an insertion heuristic for the Vehicle Routing and Scheduling Problem with Time Windows on parallel hardware. As it is experimentally demonstrated on a standard set of test problems, a good parallel implementation can produce a very substantial reduction in computation time. In particular, two procedures are described for reducing communication and idle time when the number of processors grows up.

KeyWords. Parallel Computation, Vehicle Routing, Insertion Heuristic, Transputer.

Scope and Purpose. Parallel hardware obviously has a tremendous impact in all areas related to computer science. However, that technology has not yet been fully exploited in the field of operations research. We thus describe here an implementation of an insertion heuristic for vehicle routing on parallel hardware, and show the speed-up that can be achieved with respect to computation time.

Section 1. Introduction

The Vehicle Routing and Scheduling Problem with Time Windows (VRSPTW) has recently been the focus of very intensive research and has been used to model many realistic applications [1,2]. The overall objective of that problem is to service a set of customers at minimum cost with a fleet of vehicles of finite capacity housed at a central depot. Each customer has a known demand to be serviced (either for pick-up or delivery, but not both) and a time window or time interval for that service. A time window is also included at the depot in order to define a "scheduling horizon". Each route thus starts and ends within the

bounds of that horizon which is acting as a capacity constraint (e.g. by reducing the scheduling horizon, only a few customers can be serviced by the same vehicle).

Since we consider the hard time window case, a vehicle must also wait if it arrives too early at a customer location. Accordingly, the total routing and scheduling costs include not only the total travel distance (or total travel time), but also the waiting time. Hence, spatial and temporal aspects of the problem are both very important.

Recently, we developed a new insertion heuristic for that problem which outperformed Solomon's classical heuristic for most classes of problems within his standard test set [3,4]. The aim of this paper is now to show how computation time can be improved by implementing such an algorithm on parallel hardware, as compared with the original sequential implementation.

The rest of the paper is as follows. First, Section 2 briefly describes our insertion heuristic for solving the VRPSTW problem. We then explain in section 3 how and where parallelism was introduced within the algorithm. After briefly describing the parallel architecture, we then compare in section 4 the sequential and parallel implementations, based on Solomon's test problems [4].

Section 2. An insertion heuristic for the VRPSTW

Our insertion heuristic is largely inspired by Solomon's heuristic, apart from the fact that routes are built in parallel rather than one by one. However, our heuristic introduces a new metric (the so-called generalized regret measure) in order to select the next candidate customer for insertion.

Here a set of n_r routes is initialized at once, and the remaining customers are inserted into any one of those routes, until all customers are routed (additional routes can be created if needed).

In order to get an initial estimate for n_r, we first apply Solomon's heuristic with only one initialization criterion, namely the farthest customer from the depot, and only one parameter setting $(\beta, \alpha_1, \alpha_2) = (1,1,0)$. Note that Solomon uses two different initialization criteria and four different parameter settings, and his algorithm is thus run eight times on each problem in order to produce the reported solutions (see (Solomon 87) for details). The routes are also used to select good seed customers to start our algorithm, as it is explained below.

Let $(i_{r0}, i_{r1}, i_{r2},..., i_{rm})$ be some route r with i_{r0} and i_{rm} standing for the depot. Our algorithm can now be described in the following way:

1. The seed customers are identified by selecting the farthest customer from the depot in each route generated by Solomon's algorithm (applied with only one initialization criterion and a single parameter setting). An individual route is then created for servicing each seed customer, and the remaining customers are unrouted. By using Solomon's routes, we get a good initial distribution of the n_r seed customers over the geographic area.

2. We consider the minimum insertion costs for each unrouted customer $u \in S_u$ in each one of the n_r initial routes:

   $$c_{1r}^*(i_r(u),u,j_r(u)) = \min_{p=1,...,m} [c_{1r}(i_{r_{p-1}},u,i_{r_p})], \quad r=1,...,n_r.$$

   $$c_{1r}(i_r,u,j_r) = \alpha_1 \cdot c_{11r}(i_r,u,j_r) + \alpha_2 \cdot c_{12r}(i_r,u,j_r), \quad \alpha_1+\alpha_2=1, \ \alpha_1 \geq 0, \ \alpha_2 \geq 0.$$

   where

   $$c_{11r}(i_r,u,j_r) = d_{i_r,u} + d_{u,j_r} - d_{i_r,j_r},$$

   $d_{k,l}$ = distance (in time units) between k and l.

   $$c_{12r}(i_r,u,j_r) = b_{u,j_r} - b_{j_r},$$

   b_k = current service time at k.

   $b_{k,l}$ = new service time at l, given that k is now in the route.

   Then, we select u^* such that:

   $$c_2(u^*) = \max_{u \in S_u} [c_2(u)]$$

   $$c_2(u) = \sum_{r \neq r'} [c_{1r}^*(i_r(u),u,j_r(u)) - c_{1r'}^*(i_{r'}(u),u,j_{r'}(u))],$$

   where

   $$c_{1r'}^*(i_{r'}(u),u,j_{r'}(u)) = \min_{r=1,...,n_r} [c_{1r}^*(i_r(u),u,j_r(u))]$$

   and we insert u^* between $i_{r'}(u^*)$ and $j_{r'}(u^*)$.

3. Repeat Step 2 until all customers are routed or no feasible insertion points exist for one or more unrouted customers.

We must first observe that the insertion cost formula c_{1r} is the same as the one used by Solomon, that is, a weighted sum of detour and delay, while the cost c_2 for selecting the next customer to be inserted is based on a generalized regret measure over all routes. Basically, the regret is a kind of "look ahead" that indicates what can be lost later, if a given customer is not immediately inserted within its best route. In our context, a large regret measure means that there is a large gap between the best insertion place for a customer and its best insertion places in the other routes. Hence, unrouted customers with large regrets must be considered first, since the number of interesting alternative routes for inserting them is small. On the other hand, customers with a small regret measure can be easily inserted into alternative routes without loosing much, and are thus considered later for insertion.

Broadly speaking, this new generalized measure improves upon the classical regret measure by extending the look-ahead to all available alternatives, whereas the classical heuristic is rather "short sighted" and merely looks at the best and second best alternative [5]. In our context, the generalization of the regret measure has the effect of driving problem-solving more strongly towards feasible solutions.

The parallel approach, as described above, is applied to each problem with three different parameter settings $(\alpha_1, \alpha_2) = (0.5, 0.5)$, $(0.75, 0.25)$, $(1.0, 0.0)$, and the best overall solution is chosen. It must however be noted that the algorithm is applied more than three times to each problem because we also look for potential feasible solutions with a smaller number of routes than the one given by Solomon's initial estimate.

This is done by stopping the procedure as soon as a feasible solution is found for a given number of routes (initially, Solomon's estimate) and a given parameter setting. We then remember the parameter settings that have not yet been tried for the current number of routes (if any), we decrease the number of routes by one and we repeat the whole procedure with $n_r = n_r - 1$ routes. In the process, one of the initial seed customers must be taken away, and this is done by removing the seed customer s^* which is the closest to the other ones, that is:

$$\text{remove } s^* = \arg \left[\min_{s \in S_{seed}} \{ \min_{s' \in S_{seed},\, s' \neq s} d_{s,s'} \} \right]$$

where S_{seed} is the current set of seed customers.

The number of routes is reduced in this way until we get some value $n_{r_{inf}}$ for which no feasible solution is found. At that point, we simply backtrack to $n_r = n_{r_{inf}} + 1$ routes and go on with the remaining parameter settings (if any) for that number of routes. The best solution over the three parameter settings for $n_r = n_{r_{inf}} + 1$ is then selected as the final solution.

In the very few cases where a feasible solution is not found with the initial n_r estimate, as provided by Solomon's algorithm, additional routes are incrementally added until such a feasible solution is found. When we add a new route, a new seed customer must be chosen and this is done by selecting the customer which is the farthest from the current set of seed customers, that is:

$$\text{add } u^* = \arg \left[\max_{u \in S_u} \{ \min_{s \in S_{seed}} d_{s,u} \} \right]$$

where

S_{seed} = current set of seed customers

S_u = current set of unrouted customers

With that heuristic, we were able to improve most of the best known solutions of Solomon's standard set of test problems [3]. We will now describe in the next section an efficient parallel implementation of that heuristic.

Section 3. Implementation on Parallel Hardware

Parallelization is introduced in step 2 of the above algorithm. There, minimum insertion costs are to be computed for each customer in each route in order to select the customer with the maximum generalized regret measure. Our basic approach to improve computation time is to distribute the set of unrouted customers among the available processors, so as to distribute the computation of the generalized regret measures.

Under that scenario, each processor runs the main program and individually performs step 1 of the algorithm, that is, the initialization phase where the seed customers are selected and the initial routes are created. Then, in step 2, each processor takes care of some subset of unrouted customers, based on the processor's number. If the number of unrouted customers is n_u (numbered from 1 to n_u), the number of processors is n_p (numbered from 0 to n_p-1) and the processor's

number is n_o, then processor n_o takes care of customers $n_o*(n_u/n_p)$ +1 to $(n_o+1)*(n_u/n_p)$. For example, if there are 100 unrouted customers and two processors, processor 0 would focus on the 50 first customers and processor 1 on the last 50 customers. Each processor thus computes the minimum insertion costs in each route for its subset of customers (i.e. computes an insertion cost matrix $C_p(i,j)$ where i is the ith unrouted customer assigned to processor p and j is the jth route), and identifies the customer with maximum regret in that subset.

One of the available processors, say processor 0, then runs a special program for the final customer selection. The main program running on each processor (including processor 0) sends the identity of the processor's best customer to the selection program of processor 0, as well as the associated regret measure and the best insertion place. Processor 0 then compares the regret values and identifies the best customer over all processors. The identity of that customer is then broadcasted, as well as its best insertion place, to the main program running on each processor (including processor 0). At that point, each processor individually updates the solution by inserting the selected customer at its best insertion place. Each processor also updates the insertion cost matrix accordingly.

Although simple and efficient (as demonstrated in the experiments of section 3), that approach has a few drawbacks. Those problems are described below.

3.1 Congestion for customer selection

The first problem is related to the congestion that appears at the processor running the selection program when the number of available processors grows up. This is due to the fact that all the data are sent from all processors to a single one. Moreover, depending on the topology of the network (to be discussed in the next section), the path from a given processor to the processor running the selection program can be quite long. In order to improve our results with respect to communication time, we had to distribute the program for the final customer selection. Accordingly, the selection program is now run on half of the available processors. Assuming that four processors are available and that even numbered processors (including processor 0) are running the selection program, the final selection procedure is as follows:

1) Each uneven numbered processor $n_0 + 1$ send its best customer to the associated even numbered processor n_0, that is:

   processor 1 -> processor 0
   processor 3 -> processor 2

2) Each even numbered processor n_0 then compares its selected customer with the selected customer sent by processor $n_0 + 1$ and selects the best of the two. Processor 2 then sends its best customer to processor 0:

   processor 2 -> processor 0

3) Processor 0 then compares and select the best overall customer and broadcasts the identity of that customer to all processors.

Distributing the final customer selection over many processors eliminates the congestion problem, and consequently reduces the communication time. In the following, we will refer to that procedure as the "decentralized selection" procedure.

3.2 Unequal workload among processors

Another problem is related to the fact that we want each processor to have about the same number of customers to take care of (since processors with only a few customers must wait at each iteration for the processors with many customers). Of course, the customers are well distributed at the outset, but as the algorithm goes on and customers are inserted into the routes, the number of remaining unrouted customers in each processor can be quite different (e.g. if the first customers to be inserted were all assigned to the same processor). There is thus a need to reassign the unrouted customers among the processors when such a situation occurs so as to reduce the average processor's idle time.

We first designed a very simple procedure for doing that. Given:

n_u = initial number of unrouted customers.
n_p = number of processors.

customer = vector of length n_u, where customer[i] is the ith unrouted customer.

routed = vector of length n_u, where routed[i] is true if customer i has been routed, false otherwise.

start = vector of length n_p, where start[i] indicates the starting point of processor i for scanning the *customer* vector.

sum = vector of length n_p, where sum[i] sums up the number of routed customers in the interval of customers assigned to processor i.

The procedure works as follows:

Initialization

sum[i] = 0 for i=0,..., n_p-1

start[i] = i * (n_t/n_p) for i = 0,..., n_p-1

(for the sake of simplicity, we assume here that n_p divides n_t)

At each iteration, an unrouted customer is selected and inserted in some route. Hence, the corresponding entry in the *routed* vector must be updated and set to true (which means that the corresponding customer will not be taken into account anymore by any processors). Also, the entry in the *sum* vector corresponding to the processor in charge of that customer is incremented by one.

After n_p iterations, the customers are reassigned to the processors by updating the vector *start* as follows:

```
while routed[start[0]] is true then
    start[0] = start[0] + 1;

sumall = sum[0];

for i=1 to np-1 do
    start[i] = start[i] + sumall - i;
    sumall = sumall + sum[i];

sum[i] = 0 for i =0,..., np-1;
```

This simple procedure is very interesting because it can be run locally at each processor for updating the bounds of the vector *start* and identify their new subset of customers. Unfortunately, since each processor computes the insertion cost matrix only for the subset of customers it has been initially assigned to, those insertion costs are unknown locally for any customer shifted from one processor to another during reassignment.

For example, assume that there are 2 processors and 100 customers to be routed. Initially, customers 1 to 50 are assigned to processor 0 and customers 51 to 100 are assigned to processor 1. In order to compute the generalized regret measure for each one of their customers, processors 0 and 1 must first respectively compute the insertion cost matrices $C_0(i,j)$ and $C_1(i,j)$, where i stands for the ith customer assigned either to processor 0 or 1 and j stands for the jth route. Now, suppose that the two first customers to be routed are assigned to processor 0. The reassignment algorithm would then assign customer 51 to processor 0 (start[1] being reset to 52). But, processor 0 has never computed the cost vector (51, j) since it is the first time customer 51 is assigned to it. Hence, processor 1 must send that vector to processor 0.

Consequently, the above "local" procedure was not really appropriate because some information must be transferred among processors. Therefore, a more "centralized" approach was taken in order to manage and control the transfer of information.

Basically, control is given to the processor running the final customer selection procedure (say, processor 0). At the end of the computation of the regrets, each processor now sends its identity along with the identity of its best customer (plus the regret and the best insertion place). Processor 0 then uses that information to update a vector containing the current number of unrouted customers at each processor. Using that vector, processor 0 also starts reassignment by broadcasting a message containing the identity of the "source" and "destination" processors and the number of unrouted customers to be sent from the source to the destination.

Reassignment is started based on a new "threshold" parameter. Namely, the reassignment procedure is activated if the difference between the maximum and minimum number of unrouted customers among the processors is larger or equal to the threshold.

Given a threshold T, the check is performed at processor 0 as follows. At start, a check needs to be done only after T iterations. If the difference D is below the threshold, no reassignment occurs and a new check takes place after (T - D) new iterations. Otherwise, reassignment occurs and a new check takes place after T new iterations (assuming that at start and after each reassignment, each processor gets the same number of customers). Of course, customers are transferred from the processor with the maximum number of customers to the processor with the minimum number of customers.

In the following, we will refer to that procedure as the "dynamic workload distribution" procedure. The next section will now show the impacts of the above strategies on the efficiency of our parallel algorithm.

Section 4. Computational Results

We ran a set of experiments on a Sun 3 workstation using a board with 1, 2, 4 and 6 transputers. Each transputer has 2 Megabytes of local memory and can communicate directly with up to 4 Transputers. The main host (the Sun 3 workstation) was only used in our experiments to activate the transputer networks and to read the input data files. The transputers were running under the GENESYS operating system, and the programming was done with standard Fortran. Special library routines were used for managing parallelism.

For the experiments, we used a standard grid architecture, as depicted in figure 1 for a network of 6 transputers. Note that all transputers run the main program, but only transputers 0, 2, 4 can run the program for comparing two customers and selecting the one with larger regret.

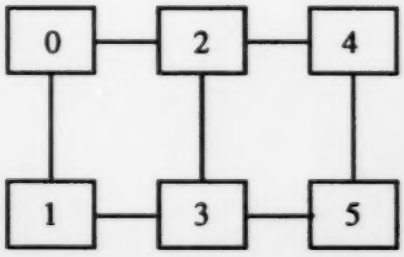


Figure 1. A grid architecture for 6 transputers

4.1 The set of problems

For the computational tests, we used the standard set of problems of Solomon [4]. The design of those test problems highlights factors that can affect the behavior of routing and scheduling heuristics, like geographical data, number of customers serviced by a vehicle and time window characteristics (e.g. percentage of time-constrained customers, tightness and positioning of the time windows, width of the time horizon). The data used for customer coordinates and demands are based on the data from the set of routing problems of Christofides et al. [6]. The geographical data were either randomly generated based on a uniform distribution (problem classes R1 and R2), clustered (problem classes C1 and C2) or semi-clustered with a mix of randomly distributed and clustered customers (problem classes RC1 and RC2). Problem classes R1, C1 and RC1 have a short scheduling horizon, and allow only a few customers per route. Conversely, problem classes R2, C2 and RC2 have a long scheduling horizon and allow a larger number of customers per route. More details about these problems, in particular details relating to the generation of the time windows, may be found in Solomon's paper [4].

All test problems are 100-customer euclidean problems and in each case an homogeneous fleet of vehicles is assumed.

4.2 The results

In Table I we show, for each class of problems, the results with the basic parallel implementation with customer selection centralized at processor 0 and no dynamic workload distribution. Results are shown for 1, 2, 4 and 6 transputers. The first number in each cell is the computation time in seconds relating to the distributed part of the algorithm, namely the computation of the generalized regret measure for the unrouted customers. The second number is the computation time for initialization and the third number is the remaining time, mainly communication and idle time. Finally, the last number is the total computation time (i.e. the sum of the three first numbers). Each number shown in the table is the average computation time at each processor taken over all processors and over all problems in the class.

	Number of Processors			
	1	2	4	6
C1	24.0	12.0	6.0	4.0
	1.4	1.4	1.4	1.4
(9 problems)	1.6	3.5	5.5	8.8
	27.0	16.9	12.9	14.2
C2	18.5	8.9	4.6	3.1
	3.9	3.8	3.9	3.9
(8 problems)	1.6	3.9	5.6	8.8
	24.0	16.6	14.1	15.8
R1	27.5	13.8	6.9	4.6
	1.3	1.3	1.3	1.3
(12 problems)	1.6	3.6	5.0	9.6
	30.4	18.7	13.2	15.5
R2	31.8	15.9	8.0	5.3
	7.4	7.5	7.5	7.5
(11 problems)	1.7	3.7	6.3	9.6
	40.9	27.1	21.8	22.4
RC1	27.3	13.6	6.9	4.6
	1.2	1.2	1.2	1.2
(8 problems)	1.6	4.8	5.5	9.9
	30.1	19.6	13.6	15.7
RC2	25.2	12.6	6.3	4.2
	5.8	5.8	5.8	5.8
(8 problems)	1.7	5.4	6.0	9.9
	32.7	23.8	18.1	19.9

Table I Computation times for 1, 2, 4 and 6 processors

As we can see from Table I, the reduction in computation time for the distributed part of the algorithm is linear with the number of processors. However, communication and idle time take a larger proportion of total computation time as the number of processors grows up. Table II thus shows the new communication and idle time,

for each class of problems, with the "decentralized selection" and "dynamic workload distribution" procedures (as described in sections 3.1 and 3.2). In the table, the first number is the computation time with both the decentralized selection and dynamic workload distribution procedures, the second number is the computation time with the decentralized selection procedure only and the number between parentheses is the computation time for the basic implementation. As in table I, each number is the average commmunication and idle time at each processor taken over all processors and over all problems in the class.

	Number of Processors		
	2	4	6
C1	2.2	3.2	-
	2.4	3.5	4.3
(9 problems)	(3.5)	(5.5)	(8.8)
C2	2.6	3.2	-
	2.9	3.7	4.5
(8 problems)	(3.9)	(5.6)	(8.8)
R1	2.1	3.3	-
	3.0	3.7	4.2
(12 problems)	(3.6)	(5.0)	(9.6)
R2	2.4	3.4	-
	2.9	4.2	5.0
(11 problems)	(3.7)	(6.3)	(9.6)
RC1	2.1	3.4	-
	3.9	4.8	4.2
(8 problems)	(4.8)	(5.5)	(9.9)
RC2	2.2	3.4	-
	4.4	5.0	4.7
(8 problems)	(5.4)	(6.0)	(9.9)

Table II Communication and idle time for 2, 4 and 6 processors

Note that the results involving the "dynamic workload distribution" procedure were obtained with a threshold value set to 2. Also, that procedure has been run with 2 and 4 processors only, because the initial number of customers is too low (100) to allow for serious problems to occur with workload distribution when a larger number of processors is used. In fact, even with 2 and 4 processors, that is 50 and 25 customers per processor initially, the improvement is not so great on average if we compare with the results achieved with decentralized selection only (apart from problem classes RC1 and RC2, where the improvement is more substantial).

There are however problems for which the dynamic workload distribution procedure was very good. We thus took, in each class, the problem whose average communication and idle time was improved the most with the addition of that procedure. The numbers in the cells of Table III have the same meaning as in Table II. However those numbers are for individual problems rather than problem classes (e.g. problem C104 is the fourth problem of class C1). Once again, those results were obtained with a threshold value set to 2.

	C104	C204	R112	R211	RC104	RC208
2 Processors	2.7 3.8 (4.8)	3.0 4.1 (4.9)	3.1 4.9 (6.4)	2.8 4.0 (5.1)	2.1 4.8 (5.4)	2.4 7.2 (8.0)

	C104	C204	R112	R208	RC107	RC208
4 Processors	3.6 4.7 (6.5)	3.5 4.7 (6.2)	5.4 7.0 (7.4)	4.7 7.1 (8.6)	3.8 5.8 (7.9)	3.5 6.6 (7.7)

Table III Communication and idle time for individual problems

Table IV now shows the final computation times with the "decentralized selection" and "dynamic workload distribution" procedures for 2 and 4 processors and the "decentralized selection" procedure only for 6 processors. The numbers in the cell of Table IV have the same meaning as in Table I, and shows a steady decrease in

total computation time as the number of processors grows up. The best results have thus been achieved with 6 processors, with a reduction in total computation time by a factor of 3 with respect to the sequential implementation for problems C1, R1, RC1, and a reduction by a factor slightly larger than 2 for problems C2, R2, RC2.

	Number of Processors			
	1	2	4	6
C1	24.0	12.0	6.0	4.0
	1.4	1.4	1.4	1.4
(9 problems)	1.6	2.2	3.2	4.3
	27.0	15.6	10.6	9.7
C2	18.5	8.9	4.6	3.1
	3.9	3.8	3.9	3.9
(8 problems)	1.6	2.6	3.2	4.5
	24.0	15.3	11.7	11.5
R1	27.5	13.8	6.9	4.6
	1.3	1.3	1.3	1.3
(12 problems)	1.6	2.1	3.3	4.2
	30.4	17.2	11.5	10.1
R2	31.8	15.9	8.0	5.3
	7.4	7.5	7.5	7.5
(11 problems)	1.7	2.4	3.4	5.0
	40.9	25.8	18.9	17.8
RC1	27.3	13.6	6.9	4.6
	1.2	1.2	1.2	1.2
(8 problems)	1.6	2.1	3.4	4.2
	30.1	16.9	11.5	10.0
RC2	25.2	12.6	6.3	4.2
	5.8	5.8	5.8	5.8
(8 problems)	1.7	2.2	3.4	4.7
	32.7	20.6	15.5	14.7

Table IV Computation times for 1, 2, 4 and 6 processors with decentralized selection and dynamic workload distribution

Overall, our computational results show that reduction in computation time is linear with the number of processors for the distributed part of the algorithm. On the other hand, the average communication and idle time, which is relatively small with 2 processors, gets much more important with 4 and 6 processors. The "decentralized selection" and "dynamic workload distribution" procedures were thus designed to improve those computation times. The results with these procedures were particularly impressive for problem classes RC1 and RC2 with 2 and 4 processors and for all classes with 6 processors.

Section 5. Conclusion

In this paper, we have described an efficient parallel implementation for a VRSPTW insertion heuristic. Since insertion heuristics are also used for solving dynamic vehicle routing and dispatching problems, where new incoming requests must be responded to in real time and incorporated into evolving routes, efficient parallel implementations are of prime importance for allowing a computer system to react quickly and properly.

Acknowledgements. We would like to thank the Natural Sciences and Engineering Research Council of Canada (NSERC) as well as the Fonds pour la Formation de Chercheurs et l'Aide à la Recherche (FCAR) of the Quebec government for their financial support.

References

[1] Bodin L., Golden B.L., Assad A. and Ball M. (1983), "Routing and Scheduling of Vehicles and Crews: The State of the Art", *Computers and Operations Research 10*, 62-212.

[2] Solomon M.M. and Desrosiers J. (1988), "Time Window Constrained Routing and Scheduling Problems", *Transportation Science 22(1)*, 1-13.

[3] Potvin J.Y., Rousseau J.M. (1991), "A Parallel Route Building Algorithm for the Vehicle Routing and Scheduling Problem with Time Windows", Publication du Centre de Recherche sur les Transports no. 729, Université de Montréal, to appear in *European Journal of Operational Research*.

[4] Solomon M.M. (1987), "Algorithms for the Vehicle Routing and Scheduling Problems with Time Window Constraints", *Operations Research 15 (2)*, 254-265.

[5] Martello S., Toth P. (1981), "An Algorithm for the Generalized Assignment Problem", in *Operational Research'81: Proceedings of the Ninth IFORS Int. Conf. on Operational Research*, J.P. Brans (Eds), North-Holland, 589-603.

[6] Christofides N., Mingozzi R. and Toth P. (1979), "The Vehicle Routing Problem", in *Combinatorial Optimization*, N. Christofides, R. Mingozzi, P. Toth and C. Sandi (Eds), John Wiley and Sons, New-York.